**JAVA 8 FEATURES IS ALSO ONE OF THE MAJOR CHANGES BROUGHT BY THE ORACLE:**

* Below are the major changes as a part of the Java 8 released by Oracle. The main concentration in the Java 8 features is to reduce the number of lines of code in order to compete with other languages like Python, R and Scala.
  + Lambda Expressions.
  + Functional Interfaces.
  + Default and static methods in Interfaces.
  + Predefined Functional Interfaces:
    - Function
    - Predicate
    - Consumer
    - Supplier.
  + Streams
  + Double Colon Operator (::) for Method and Constructor Reference.
  + Date and Time API.
  + Optional Class.
  + Nashron Javascript Engine.
* **LAMBDA EXPRESSIONS:**
  + The main advantage of lambda expression is to bring the benefits of the Functional Programming to Java.
  + In simple Lambda expression is an anonymous function.
    - Function without name
    - Function without return type.
    - Function without modifiers.
  + Below is the way how to write Lambda Expression:
    - Normal Method:

|  |
| --- |
| **public void print(){**  **System.out.println(“Hello”);**  **}** |

* Below is how we can write Lambda Expression for above method:

|  |
| --- |
| **() -> System.out.println(“Hello);** |

* If there is only one line of code then curly braces are optional for the Lambda Expression.

|  |
| --- |
| **Example 2:**  public void add(int a, int b) {  System.out.println(a+b);  }  **Below with Lambda Expression:**  (a,b) - > System.out.println(a+b);  In the above scenario Compiler is going to guess the parameter type automatically. Hence we don’t need to specify. Since only one line of code Parenthesis are optional. |

|  |
| --- |
| **Example 3:**  public int squareIt(int n){  return n\*n;  }  **Below with Lambda Experession:**  (n) -> {return n\*n;} or  n ->n\*n;  Parameter type is removed since Compiler is going to guess automatically. If only one line of code after Lambda Expression no need of Parenthesis. Return statement is mandatory inside Parenthesis otherwise outside its not mandatory. Return type will also be guessed by Compiler automatically.   1. **With Curly braces if we want to return something then return statement is mandatory.** 2. **Without curly braces if we want to return something then return statement should be avoided.** |

* An Interface which has only one abstract method irrespective of how many static and default methods in that interface is called Function Interface. Function Interfaces are used for calling Lambda Expressions. Below are some examples for the Functional Interfaces:
  + - **Runnable.**
    - **Callable.**
    - **Comparable.**
    - **EventListener.**
    - **Comparator.**
* To denote an interface as a Functional Interface then we need to use @FunctionalInterface on top of the particular interface. But this annotation is optional to specify.
* Where ever functional interfaces are there, there only we can use Lambda Expressions other places we cannot use. Hence it is very specific concept.
* Lambda Expression can be applicable for not only Predefined class objects, it can be applicable for user defined class objects. Please refer LambdaExpressionWithUserDefinedObjects.java.
* In Collections where ever we need sorting, we mostly use the Functional Interfaces Comparable and Comparator which has compareTo() and compare() methods respectively. If we need ascending order or descending order then we can use Lambda Expressions on the Functional Interfaces Comparator and Comparable interfaces which has only one abstract method.

**ANONYMOUS INNER CLASSES VS LAMBDA EXPRESSIONS:**

* Between the above two concepts Anonymous inner classes is very powerful which can be applicable for implementing or extending any class by an anonymous class (Class without name).
* But for Lambda Expressions can be applicable only for the Anonymous inner classes that implements only Functional Interfaces i.e. an interface with only one abstract method.
* We can define a Thread with Anonymous Inner classes extending Thread class and implementing Runnable interface.
* **Please refer AnonymousClassExtendsThread.java and AnonymousClassImplementsRunnable.java.**

**DEFAULT AND STATIC METHODS INSIDE THE INTERFACE:**

* Default Method: This is very important concept came in Java 8. Till Java 1.7 all the methods inside the interface are by default public and abstract.
* From Java 8 we can write the default methods inside the Interface.
* What is the necessity of default methods in interface in java.
* If we don’t want the implementation classes of the interface to be affected because of the addition of another abstract method i.e. if we introduce a new abstract method then all the implementation classes have to provide the implementation for that newly added method in the interface for new functionality.
* Default methods are also known as Virtual Extension Method or Defender Method.
* **For** Example in Collections Framework, Collection interface is implemented by almost many classes like ArrayList, LinkedList, Vector, Stack, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque etc. Suppose if we want to add any new functionality to Collection interface, if I add any abstract method then all the above implementation classes will get impacted. So in order to avoid this, new concept called default methods came in interface which will have default implementation in interface, implementation classes no need to provide implementation for default methods. If we are not satisfied, then we can provide the implementation for default methods in our implementation class.
* **\*\*\*\* We can have default keyword used for methods only inside the interface, we cannot use the same in classes as default have different meaning inside classes.**
* **\*\*\*\* Object class methods we should not provide as default methods in interface as by default all the Object class methods are available to each and every class. It is not required again to have the same in the interface as default methods.**
* **\*\*\*\* Java won’t provide any support for multiple inheritance because of the ambiguity problem w.r.t classes.**
* **\*\*\*\* If two interfaces have same default methods with same signature and our class implements those interfaces then we should provide the implementation for that common default method in our class otherwise ambiguity will occur.**
* **By default, all the default methods are available to implementation class.**
* **Default method can be called only with the object of the implementation class.**

**STATIC METHODS IN INTERFACE:**

* In Java 1.8 one more beautiful feature in interface is, providing the flexibility to write static methods. The main advantage of writing the static methods in the interface is code reusability. For example if we have one method in interface A as we can achieve multiple inheritance in Java with interfaces we can inherit the any number of interfaces in other interface.
* The main application of static methods in interfaces is writing the utility methods in interface instead of class as cost of creation of class is more than the cost of creation of interface.
* From Java 8 we can write main method in interface also and can run the same without any issues in eclipse.
* By default, all the static methods inside the interface will not be available for the implementation classes. We can only call them only with the interface name from other places of the application.
* We can have any number of static methods in an interface. Please Java 8 features application in your program.
* ***Whether a class is implementation class of the interface or not, static methods inside the interface can be called only with interface name.***

**PREDEFINED FUNCTIONAL INTERFACES IN JAVA 8:**

* **There are many predefined functional interfaces in java. util. function.\* package which has predefined functionality. Below are few of them:**
  + **Predicate.**
  + **Function.**
  + **Consumer.**
  + **Supplier.**
* **Two Arguments Predefined Functional Interfaces i.e. Functional interface that takes exactly two parameters.**
  + **BiPredicate.**
  + **BiFunction.**
  + **BiConsumer.**
* **Primitive Predefined Functional Interfaces:**
  + **IntPredicate.**
  + **IntFunction.**
  + **IntConsumer.**
  + **IntSupplier.**
* **There are many other Predefined Functional Interfaces in the java.util.function.\* package.**
* **Predicate FI:** The main usage of the Predicate FI is for conditional checks in java. For example given number is even or not, given String has length >5 or not etc. We have if else also in java for conditional checks, but as we can use Lambda expressions with FI we can write very much concise code. Please refer PredicateFI.java.
* We have only one abstract method in Predicate i.e. test (T t) which accepts any type and returns Boolean.
* In Predicate we have to specify always only the input type as input type may change always. And we don’t need to specify the output type as it is boolean always for all Predicates.
* We can have multiple logics also evaluated with and, or, negate functions which are defined in Predicate.
* **Function FI:** Function is also predefined Functional Interface which is mainly used for performing some business logic i.e. if we provide some input process the input and give us the output.
* **The method available in Function FI is R apply( T t);**
* **We have another two important methods in Function interface that has below functionality.**
  + **f1.andThen(f2).apply(10);** First f1 will be evaluated and then result of f1 will be used in f2 and the function f2 is evaluated.
  + **f1.compose(f2).apply(10);** First f2 will be evaluated then f1 will be evaluated.
* **Consumer FI: Consumer always take some input and does not return any value.**
* The available method in Consumer Functional Interface is **void accept(T t);**
* The above will only take some input and perform some operation and does not return anything. For Example it does some DB operation and does not return any value.
* Chaining in Consumer interface is possible with the andThen() same as in the Function FI.
* **Supplier FI:** If there is a requirement for us where we don’t need to provide any input and we need some object as output. Then we can for the Supplier FI.
* **The method available in the Supplier FI is R get(); There is no input to the get() and we will get only output.**

**TWO ARGUMENTS PREDEFINED FI:**

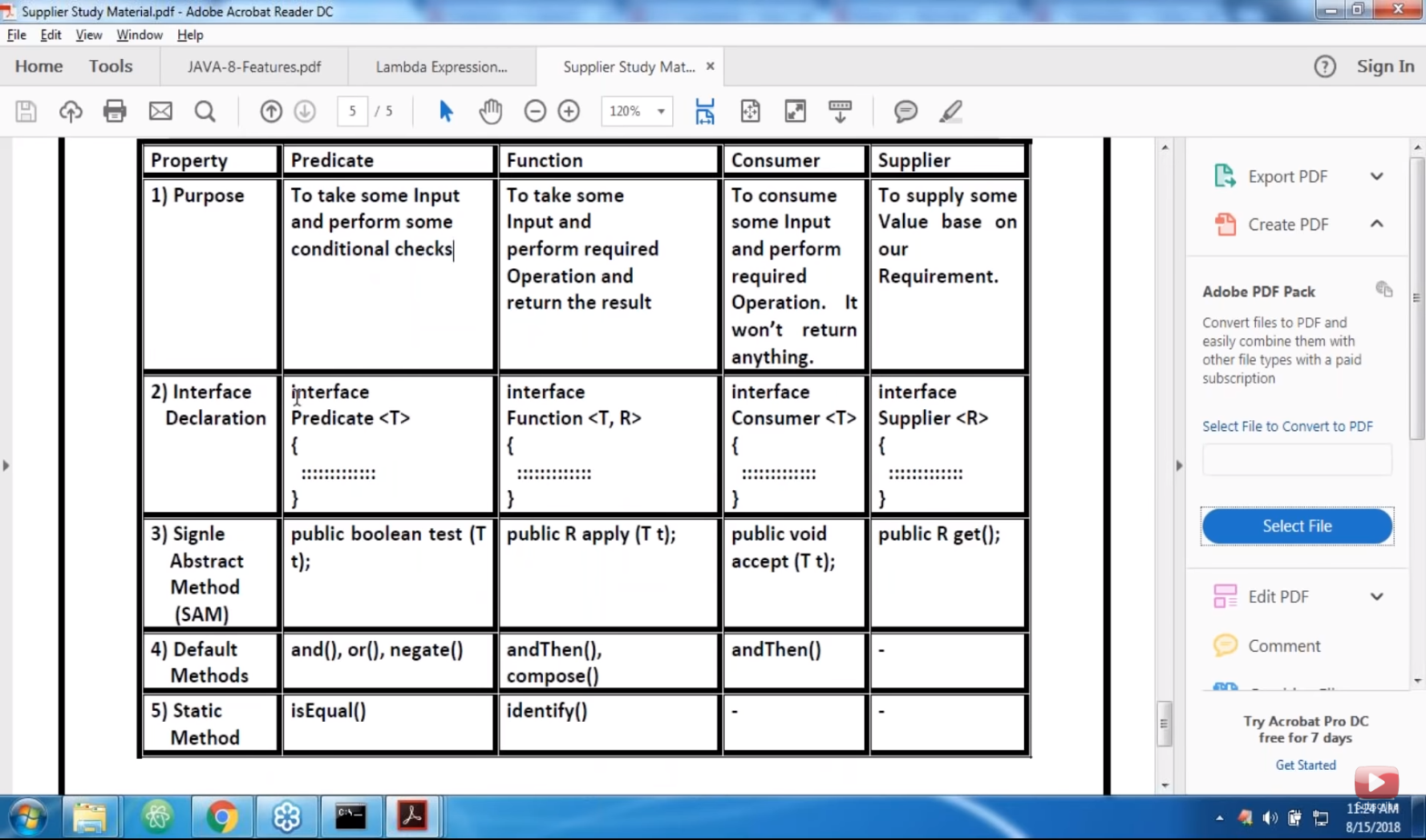
* Out of four in above predefined FI only three accepts inputs as one parameter (Predicate, Function and Consumer). Suppose my requirement is that I need to have two parameters as input and need to perform some conditional check. Then we need to go for **BiPredicate**.
* The same way we have **BiFunction** which takes two arguments as input performs some operation and returns output.
* The last one we have is **BiConsumer** which takes two input parameters and does not return anything.
* Except they take two arguments all the others functionalities will remain as is.

**PRIMITIVE PREDEFINED FUNCTIONAL INTERFACES:**

* As we know all the above FI takes only reference type as input, even though we pass primitive as input it will internally convert to Wrapper class objects and to primitive for operation again. We can observe that there are lot of conversions happening which will impact the performance severely. Hence Primitive Predefined FI are introduced which takes primitive as input and does not do any conversions.
* Below are some primitive predefined FI:
  + IntPredicate.
  + DoublePredicate.
  + LongPredicate.
  + IntFunction.
  + DoubleFunction.
  + LongFunction.
  + IntToDoubleFunction.
  + IntToLongFunction.
  + LongToIntFunction.
  + LongToDoubleFunction.
  + DoubleToIntFunction.
  + DoubleToLongFunction.
  + ToIntFunction.
  + ToDoubleFunction.
  + ToLongFunction.
  + ToIntBiFunction.
  + ToDoubleBiFunction.
  + ToLongBiFunction.
  + IntConsumer.
  + LongConsumer.
  + DoubleConsumer.
  + ObjDoubleConsumer.
  + ObjIntConsumer.
  + ObjLongConsumer.
  + BooleanSupplier. –getAsBoolean()
  + LongSupplier. –getAsLong()
  + IntSupplier.—getAsInt()
  + DoubleSupplier.—getAsDouble()
* Please refer the programs for more information.
* Suppose if both input and output are of same type then go for the UnaryOperator. UnaryOperator is child for the Function FI.
* We have primitive types also in UnaryOperator as below:
  + IntUnaryOperator. –applyAsInt()
  + DoubleUnaryOperator.—applyAsDouble()
  + LongUnaryOperator.—applyAsLong()
* Suppose if both inputs are same and output is also same then we can go for the BinaryOperator which is child of BiFunction.
* We have primitive types also in the BinaryOperator.
  + IntBinaryOperator.—applyAsInt()
  + DoubleBinaryOperator. –applyAsDouble()
  + LongBinaryOperator. –applyAsLong()

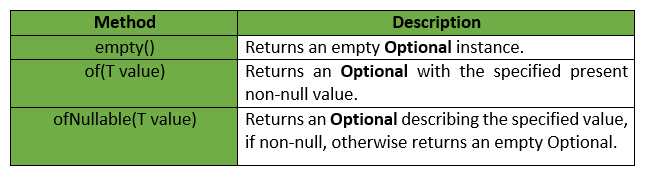
**METHOD AND CONSTRUCTOR REFERENCE IN JAVA 8:**

* We can use :: symbol for method and constructor reference in java.
* Method reference is an alternative to the lambda expression and the advantage of the Method reference is code reusability.
* Suppose whatever the logic we are going to write in Lambda Expression if it is already available, we can refer the method with the Method Reference so that we can avoid writing the code again.
* Method Reference for static method Classname :: methodname.
* Method Reference for instance method Class reference :: methodname.
* When we are using the method reference number of arguments to the method which we are referring and our method should be same otherwise we will get CTE. There is no restrictions for the method name and the access modifier.



**Optional Class in Java:**

* Optional class in java is very useful to avoid a greater number of null checks in our application when we are writing the code.
* Optional class is present in java.util package and is introduced in java 8.
* To avoid NullPointerException usually we will add a greater number of null checks which makes code less readable. With the Optional class came into picture we can write very clean code with the Optional class methods.
* All the objects of the Optional class are final and immutable.
* Below are the important static methods in the Optional class.
  + of(T value) --- This method creates an Optional object with the specified value if non-null, if value is null NullPointerException will be thrown.
  + ofNullable(T value) --- This method creates an Optional class object with the specified value if it is not null, otherwise it will return empty Optional class object.
  + empty() --- This method returns the empty Optional class object.



* Below are the instance methods available in the Optional class:
  + isPresent() --- this method returns true if the Optional container is not empty and has some values otherwise it returns false.
  + ifPresent() --- this method’s Consumer lambda expression will be executed if the Optional Container has some elements otherwise it will not return anything.
  + orElse() --- this method will be executed if the Optional container is empty and does not have any elements. This method is mainly for returning the default values if the Container is empty.
  + orElseGet() --- this method will be executed if the Optional container is empty and does not have any elements. This method’s Supplier Lambda expression will be executed and if the value is present it will be returned otherwise we will get the output of the Supplier expression.
  + orElseThrow() --- This method throws the exception as new Exception() if the values are not present in the container.
  + hashCode() ----This method returns hashCode for the value in the Optional Container otherwise if the Container is empty 0 will be returned.
  + toString()—To String representation of the Optional object is returned.
  + filter() ---filter method returns the Optional Container with the value that met the condition otherwise it returns empty Optional Container.
  + map() --- This method will be executed if the Container is having some values and Optional object will be returned with result if the condition is met otherwise we will get empty Optional Object.
  + get() --- returns the value in the Optional container, if the value not exists then it will throw NoSuchElementException.
  + equals() --- This method returns true if the two Optional Containers have same elements.

**STREAMS IN JAVA 8:**

* Streams is very important concept that is used with Collections.
* Collections framework stores the group of objects.
* Stream API is used to manipulate the group of Objects in the Collections or any group of elements we can manipulate with the Stream.
* Streams has many important methods that can be operated on the group of objects.
* Below are some the important methods in Stream API.
  + stream() ---this method is used to return the stream object for a particular collection as list.stream().
  + filter()—with the stream object we can invoke the filter() if we want to filter the elements in the collection based on the some condition for example filtering all the even numbers from the list of numbers as below. list.stream().filter(i -> i%2==0).collect(Collectors.toList()); The collect() is for the purpose of collecting all the elements to a list that has met the condition.
  + map() --- This method is applicable for the scenarios where we can give some input and that has to be applied to all the elements in the collection as list.stream().map(I ->I\*2).collect(Collectors.toList());
  + forEach()—This method performs some operation on each and every element in the list and does not return anything.
  + count() --- This method returns the number of elements that has met the specified condition as list.stream().filter(I -> I%2==0).count(); This returns the even number count.
  + sorted()—This method sorts the elements based on the default natural sorting order.
  + sorted(Comparator c) --- This method sorts the elements based on the specified Comparator.
  + min(Comparator c) --- This method returns the first element in the list after sorting based on the specified comparator.
  + max(Comparator c) –This method returns the last element in the collection after sorting based on the specified comparator.
  + toArray() – This method returns the collection in the array format so that accessing from the array is easier.
  + Stream.of(Group of Elements) This method takes any group of elements and returns the Stream so that we can perform all the above operations. And we have IntStream, DoubleStream, LongStream for the primitives we can use.
  + findFirst() ---Always returns the first Element in the stream.
  + findAny() –Can return any element in the stream, but for non-parallel stream it returns mostly the first element in the stream but not always. In parallel stream the output is non-deterministic.
  + parallelStream() – Its mainly used if we are processing huge data, it creates multiple threads internally and increases the performance.
  + allMatch( Predicate p) – This method takes predicate as input returns true if all the elements in the stream satisfies the given predicate, otherwise it returns false.
  + anyMatch(Predicate p) – This method takes Predicate as input and returns true if any of the elements in the Stream matches with the given Predicate otherwise returns false.
  + noneMatch(Predicate p) – This method also takes Predicate as input, if none of the elements in the Stream matches the Predicate then this method returns true else returns false.
  + **The above three match methods are very much useful for conditional checks.**
  + distinct() – returns the Stream of unique elements by removing duplicates.
  + limit() – limits the number of elements that has to be shown in the output.
  + skip(long l) –skips the number of elements specified.

**\*\*\*\*\*** Below are all the ways how we can create Streams in Java 8:

* list.stream() --- returns stream object
* Arrays.stream() ---returns stream object.
* Stream.of() ---returns Stream object.
* Files.lines(Paths.get(“test.txt”); --- returns the lines from the text file as Stream.
* For integer stream we get like Integer.of(10, 30, 23, 40);
* There are many important methods in the IntStream class like range(), sum(), average(), skip(), min(), max(), count(),summaryStatistics() which returns IntSummarStatistics object which contains all the information about the elements that are streamed like sum, average, min, max ,count of all stream elements etc.

Below are more details about the IntStream methods:

* range()---- returns all the elements in elements in the specified range like IntStream.range(1, 10) – returns all elements from 1 to 9.
* sum() --- adds all the elements in the Stream specified.
* average() ---calculates the average() of all the elements in the Stream and returns the value in the OptionalDouble Container.
* min() – returns the min value in the IntStream container.
* max() –returns the max value in the IntStream container.
* skip() –skips the specified no of elements from the Stream like IntStream.range(1, 10).skip(5)—skips first 5 elements and returns 6,7,8,9.
* count() – returns the number of elements in the Stream.

Difference between map() and flatMap() in the Stream API:

* map() ----The function inside the map() takes one element as input and returns one element as output. For Example I have stream of strings like Stream.of(“Ravi”, “Kiran”, “Dhyan”). We will pass as input for map() and get Stream of Integers like we can get the length of each string. Input is Stream of Strings and Output is Stream of Integers
* flatMap() --- This method has two functionalities like one is the functionality done by map() and flattens the Stream. This method takes one element as input and returns Stream of elements. For example we have Stream of List of Integers and flatMap() takes one list as input returns the stream of stream of integers and flatMap() flattens all the elements in one list.

**Parallel Streams:**

* Parallel Streams are very useful to increase the performance of java code when are dealing with the large collections.
* Parallel Streams use the multi core architecture to enhance the performance.
* Parallel Streams should be used only when we have below requirements:
  + Stateless: The state of one operation should not affect other operation.
  + Association: The result of the operation should not be impacted by the order of operands.
  + Non-interfering: The operation should not impact the data source.
* If we use Parallel Streams wrongly below are the disadvantages:
  + Inter-thread communication is always dangerous and may take always time.
  + Each sub-stream is a thread and acting on the data, it may have more overhead when compared to sequential streams.’
* Below are two ways how we can create Parallel Streams:
  + parallelStream() --- We can get parallel Stream by using this method on the collections.
  + parallel() --- To get a parallel stream from the sequential stream.
* Below is the Architecture how sequential stream and parallel stream works:



* In the above picture we can see that parallel streams and will be executed in multiple cores and combines the results after processing. Suppose we have a scenario like we have n number of employees and we want find a count whose salary is >12000. Sequential streams will process element by element and takes time for calculating. Parallel streams help in achieving the results faster.

**Where to use Parallel Streams:**

* PS has to be used where ever the output is not dependent on the order of the elements in the collection.
* PS should be used when there are high performance issues with sequential streams.
* PS should be used mainly with the aggregate functions.
* PS can quickly iterate over the large sized collections.
* If the environment is not multi-threaded, then Parallel Stream creates thread and can affect the new requests coming in.